**AMENDMENTS TO THE SPECIFICATION:**

Please replace the paragraph beginning at page 1, line 5, with the following rewritten paragraph:

A1

--The present invention relates generally to the reuse of software code within the area of ~~objected~~ object-oriented programming languages, and more specifically to a system constructed in accordance with a programming language where software reusability is achieved by the use of *reusable software units* (or *connectons*), defined modularly and hierarchically, and by the ~~remotion~~ removal of all absolute addresses from *connectons*.--

Please replace the paragraph beginning at page 1, line 20, with the following rewritten paragraph:

A2

--A programming language is object oriented if it supports objects as a language feature, and objects belong to classes that can be incrementally modified through inheritance (i.e., Simula, Smalltalk, C++, Eiffel and Ada). The essence of object oriented programming is the hiding or encapsulation of the "inner" state of entities and the specification of interactive properties of entities by an interface of operations (the events in which they may participate).--

Please replace the paragraph beginning at page 3, line 22, with the following rewritten paragraph:

A3

--Pipes, ~~through~~ despite their simplicity, were one of the first constructs to provide software reuse. Pipes used in the UNIX operating systems provide an interesting mechanism for software reuse. Programs in the operating system can behave independently sending their outputs through a pipe to another program. Pipes have severe limitations, however. They can pass only unstructured data like characters, and they provide only a one-direction mechanism.--

Please replace the paragraph beginning at page 7, line 7, with the following rewritten paragraph:

A4

--Axioms III and IV have been partially removed by the use of adapters. Instead of sending messages directly to an object, messages are sent to an intermediate object known as the adapter that converts one object interface to another. Facade systems based on graphical systems give the false impression that axioms III and IV have been removed. However, the underlying mechanism is so complex that the best that can be expected is that the components existing in the library are enough to build an application, otherwise lots of difficult tricks must be mastered to complete the programming task, see *Extend User's Manual. Imagine That Inc., 1997.* ~~**Error! Reference source not found.**~~].--

Please replace the paragraph beginning at page 9, line 8, with the following rewritten paragraph:

A5

--Systems (within the art of System's Theory) were designed to represent asynchronous communications, like those found in simulation frameworks ~~**Error! Reference source not found.**~~, see *B.P. Zeigler. Theory of Modeling and Simulation. John Wiley, 1976*, and thus systems have only been provided with a one way communication mechanism. This simple fact prevents systems theory from being used as a general framework for programming. Although in principle, a synchronous system can be represented by an equivalent asynchronous system, this transformation would lead to cumbersome specifications, because all synchronous communications should be represented by two one-way links, thus doubling the interface of any synchronous system.--

Please replace the paragraph beginning at page 12, line 3, with the following rewritten paragraph:

A6

--Figure 1 represents a ~~A~~ possible connection among three connectons A 102, B 104 and C ~~of~~ 106 according to the present invention ~~is represented in Fig. 1~~. Connecton A is linked to connecton B through a channel (link) 108 starting at gate **go** 112 of connecton A, and ending at gate **start** 114 of connecton B. Connecton A is linked to connecton C through a channel 110 from gate **go** 112 to gate **begin** 118. Connecton A

is linked to connectons B and C. However, there could be more linked connectons. Connecton A could be linked just to one connecton, or it could be linked to no connecton at all. For example, connecton B has a gate **stop** 116 and connecton has a gate **break** 120, which gates are linked to no other connecton. A channel, or link, makes the interface between two different gates, thus a connecton only needs to know its own output gates. The name of the gates of outside connectons, that represent message names, are entirely handled by the channel support, permitting removal of Axiom 4. It may look strange that although channels represent a bi-directional communication link, they are represented by an arrow. The arrow represents the direction of the request, the requested information travels is in the opposite direction and thus is not explicitly represented.--

A6

Please replace the paragraph beginning at page 13, line 3, with the following rewritten paragraph:

--The function $a_g$ on input gate $g$ of signature $(I_g,O_g)$ is expressed by

$$a_g: Q \times I_g \rightarrow Q \times O_g.$$

An action ~~correspond~~ corresponds to a method in the object paradigm. An action $a_g$ receives input values from $(Q \times I_g)$, produces a change in the connecton state, and returns a value from $O_g$. As a side effect, an action on a connecton can trigger other actions on the connectons linked to it. An action can trigger state changes in other connectons, but these changes are made by actions defined on those connectons, so modularity is not jeopardized. We do not attempt to make a formal definition of this side effect for it will be clear in the next examples.--

A7

Please replace the paragraph beginning at page 14, line 22, with the following rewritten paragraph:

-- Fig. 2 represents an OR connecton 202. This connecton has one input gate called **out** 204, and two output gates called **in1** 206 and **in2** 208. The input ~~gates~~ gate corresponds to an action ~~actions~~ (~~methods~~ method) defined in the connecton. The output gates correspond to request for actions (message calls) that a connecton can send

A8

to its peers. The OR connecton has no internal state and computes the logical Or of the values present at output gates **in1** and **in2**. This connecton is described by

$$\mathrm{OR} = (\{\mathbf{out}\},\{(\varnothing,\mathbf{B})\},\{a_{\mathrm{out}}\},\{(\mathbf{in1},\mathbf{in2}\}),\{\varnothing,\mathbf{B}),(\varnothing,\mathbf{B})\})$$

where **B** = {0,1} is the set of Boolean values, and ∅ represents that no value is needed. The output function is omitted for no adaptation is needed.--

A8

Please replace the paragraph beginning at page 15, line 25, with the following rewritten paragraph:

--We describe now, in more detail, the bank account example used in a previous section. Fig. 3 represents the client connecton 302 that receives in the output gate **deposit:** 304 the values deposited in bank accounts B1, B2 and B3 at respective connectons 308, 310, 312 The connectons 308, 310, 312 have respective value gates 314, 316 and 318 that are linked to the client connecton via respective channels or links 320, 322, 324. The client is described by

$$C = (\{\mathbf{value}\},\{(\varnothing,\mathbf{R})\},\{a_{\mathbf{value}}\},\{\mathbf{deposit:}\},\{(\mathbf{Names},\mathbf{R})\},\{\mathrm{ouputFunction}_{\mathbf{deposit:}}\})$$

where **Names** is the set of client names (not defined here for simplicity).--

A9

Please replace the paragraph beginning at page 15, line 31, with the following rewritten paragraph:

--Gate **deposit**: is attached to an output function described in Smalltalk by the block: **[:x| x inject: 0 into: [:a :b| a + b]]**. This block converts the input list into a single value (the sum of values in the list). This function also provides the default value 0 if no channel is connected to gate **deposit**:. The action associated with input gate **value** 306 is defined by

**value**

^***out*** deposit: myself

where *myself* represents the bank customer.--

A10

Please replace the paragraph beginning at page 17, line 18, with the following rewritten paragraph:

-- Fig. 4 represents an ensemble with three connectons 202, 404 and 406, and the executive 408. The ensemble is defined by

A11

$$M_{\mathrm{ORTest}} = (inGates, \{inSign_g\}, \varepsilon, M_\varepsilon)$$

where

$$inGates = \{\mathbf{in1}:, \mathbf{in2}:, \mathbf{out}\}$$
$$inSign = \{(\mathbf{B}, \varnothing), (\mathbf{B}, \varnothing), (\varnothing, \mathbf{B})\}$$
$$M_\varepsilon = (\{q_{0,\varepsilon}\}, q_{0,\varepsilon}, \sigma, \Sigma^*)$$.--

Please replace the paragraph beginning at page 17, line 25, with the following rewritten paragraph:

--The executive has just one state that corresponds to a single ensemble structure defined by

A12

$$\sigma(q_{0,\varepsilon}) = (C, \{M_c\}, L)$$

where

$$C = \{\mathrm{H1}, \mathrm{H2}, \mathrm{OR}\}$$
$$\{M_c\} = \{M_{\mathrm{H1}}, M_{\mathrm{H2}}, M_{\mathrm{OR}}\}$$
$$L = \{((\mathrm{ORTest}, \mathbf{in1}:), (\mathrm{H1}, \mathbf{value}:)), ((\mathrm{ORTest}, \mathbf{in2}:), (\mathrm{H2}, \mathbf{value}:)), ((\mathrm{OR}, \mathbf{in1}), (\mathrm{H1}, \mathbf{value})), (\mathrm{OR}, \mathbf{in2}), (\mathrm{H2}, \mathbf{value})), ((\mathrm{ORTest}, \mathbf{out}), (\mathrm{OR}, \mathbf{out}))\}$$.--

Please replace the paragraph beginning at page 18, line 1, with the following rewritten paragraph:

--The following example explains how connectons may be used within a program. First, we define a "holder" connecton. A holder connecton has two input gates: 1) **value**: that sets a state variable to a given value storing this value, 2) **value** that returns the stored value. The connecton ensemble 402, represented in Fig. 4, is composed of one OR connecton 202, linked to holder connectons, H1 404 and H2 406. The ensemble has two input gates **in1**: 410 and **in2**: 412 to set logical values, and the gate **out** 414 to communicate the result.--

A13

Please replace the paragraph beginning at page 19, line 1, with the following rewritten paragraph:

--Modularity and hierarchy connecton building can be used as a paradigm to represent very complex systems. Fig. 5 illustrates how "small" connecton granularity may be. We model the factorial function by a connecton. In Fig. 5, a connecton 502 receives an initial value n at gate **fact:** 510 for calculating a factorial. The value n is then transmitted to gate **fact:** 506 of a factorial connecton 504, which determines the factorial of the initial value n by recursively transmitting values associated with the factorial from gate **out:** 508 to gate **fact:** 506 of the factorial connecton 504. The recursive definition of factorial is given by

A14

**fact(n)**

```
    if n = 0
        1
    else
        n × fact(n-1)
```

The connecton representation of this function for connecton 504 is given by

**fact: n**

```
        n = 0 ifTrue: [^1].
        ^n * (out out:(n-1)).--
```

Please replace the paragraph beginning at page 19, line 22, with the following rewritten paragraph:

-- Fig. 6 shows the Node connecton 602 that serves as a basis to build the Erastosthenes sieve of prime numbers. In Fig. 6, the connecton 602 holds a number **id** 604 and receives an incoming number via gate **in:** 606 and transmits an outgoing number via the gate **out:** 608. Fig. 7 represents the Erastosthenes sieve connecton 702 initial configuration, having an executive connecton 704. In Fig. 7, the sieve connecton 702 receives a number via gate **in:** 708, and the executive connecton 704 also recives a number via gate **in:** 706. Each connecton 804-808 of Erastosthenes sieve connecton 702 in Fig. 8 is based on connecton 602 in Fig. 6 and holds a different prime number named **id** (not shown here for simplicity). Each connecton 804-808 tests if it can divide an incoming number received at gate **in:** by **id**. If the ~~node~~ connecton can divide the incoming number then the number is discarded for it is not prime. If the number cannot be divided by id, the ~~node~~ connecton sends the number to other connectons, which ~~performs~~ perform the same test, and so on. If the number cannot be divided by the last

A15

A15

connecton 808 in the chain, the ~~element~~ connecton sends a signal to the executive connecton 704 to create a new ~~node~~ connecton to handle the found prime. This action is defined by

**in: aNumber**

```
(aNumber \\ id = 0) ifTrue: [^nil]. "Tests if aNumber is divisible by id"
^out out: aNumber     "Sends the number to the next node for checking."--
```

Please replace the paragraph beginning at page 20, line 17, with the following rewritten paragraph:

A16

-- The structure of the pipeline after receiving the sequence <2,3,4,5,6,> is represented in Fig. 8, where there is a node for primes 2, 3 and 5, represented by nodes (connectons) N2 804, N3 806 and N5 808, respectively. Initially, the network contains just the executive connecton 704 connected to gate **in:** 708 as depicted in Fig. 7. When the first number arrives the executive connecton creates the first ~~node~~ connecton 804, disconnects itself from the network (i.e., disconnecting itself from gate **in:** 708) and connects the created ~~node~~ connecton to the network (i.e., connecting gate **in:** 606 of connecton 804 to gate **in:** 708 of connecton 702) and to itself (i.e., connecting gate **out:** 608 of connecton 804 to gate **in:** 706 of connecton 704). The executive connecton becomes the last connecton. The other depicted connectons 806 and 808 are added in the same fashion. The action **in: aNumber** handles the arrival of a prime number to the executive connecton and it is defined by

**!Pipeline Interface method!**

**in: aNumber**

```
|n new gate|
new := ('N',aNumber asString) asSymbol. "Node name: #N<aNumber>"
n := self add: new class: Node. "creates a new nod""
n id: aNumber. "Sets the id of the new node"
last = #Network ifTrue: [gate := #in:]
ifFalse: [gate := #out:]. "Sets the gate variable"
"Unlinks the last node (or the network)"
self unLink: last port: gate from: #Executive port:#in:.
"Links the last node to the new node"
self link: last port: gate to: new port:#in:.
"Links the new node to the executive"
self link: new port: #out: to: #Executive port: #in:.
```

A16

last := new. "The new node becomes the last node"
^aNumber.--

Please replace the paragraph beginning at page 21, line 2, with the following rewritten paragraph:

A17

--DESMOS, the embodiment of the invention implementing the connecton paradigm as disclosed herein is built on the Smalltalk language, see *Smalltalk MT User's Guide*. Object Connect, 1995. Smalltalk was used for simplicity of the delegation implementation. However, the DESMOS implementation of the connecton paradigm of this invention is not limited to Smalltalk, but to any object-oriented programming (OOP) language that can use delegation and reflection. The reader should note that those skilled in the art will understand how and which changes to make in conventional compilers/interpreters of existing languages in order to implement an embodiment of this invention.--

Please replace the paragraph beginning at page 21, line 11, with the following rewritten paragraph:

A18

--Fig. 9 represents a block diagram of the Connecton Kernel 902. For simplicity, we have only represented a connecton ensemble 904 with its executive 906 and with just one connecton 908 in Fig. 9. The information about channels is only depicted for connecton C 908. The main difference between the abstract description of ensembles and the actual implementation is that connectons keep their own channel information. This option is more efficient and can become easier to implement in parallel/distributed computers.

Each connecton or connecton ensemble has a reference to the ensemble within which it belongs 914, 924. This reference, however, is **nil** if the connecton is the topmost one. The ensemble connecton also has a reference to its executive 912. Additional variables are currently used to manage the link information: *extChannels* 928 is an association list where each entry has the format

$$gate \rightarrow ((connecton_1, gate_1, fFilter_1, rFilter_1),$$
$$(connecton_2, gate_2, fFilter_2, rFilter_2),$$

...

$(connecton_{n},gate_{n}fFilter_{n},rFilter_{n}))$

A18

where $gate_i$ is an input gate if connecton $i$ is not the network, and is a network output gate otherwise. The variable ~~intChannel~~ *intChannels* 918 is associated with the network and defines all the channels starting at the network input gates. The two variables are defined for implementation convenience: *revExtChannels* 930 and *revIntChannels* 922. The reverse channels are maintained to make the operation of removing connectons in run-time, faster. There is a reverse channel for each (direct) channel in the model.--

---

Please replace the paragraph beginning at page 22, line 10, with the following rewritten paragraph:

---

A19

--Variables ~~extLinks~~ *extChannels* 920 and ~~intLinks~~ *intChannels* 918 are instances of class **MessageTable**, that implements a mapping table for messages. Each message is an instance of class **SCMessage** and is defined by 6 variables:

*m_sender*, the connecton that sends the message
*m_receiver*, the connecton that will receive the message;
*m_selector*, the message name;
*m_arguments*, the message arguments;
*fFilters*, channel direct (forward) filter; and
*rFilters*, channel reverse filter.

The **SCMessage** method **value** is defined by

**value**

```
m_receiver _sender: m_sender.
(fFilter == Identity) & (rFilter ==Identity) ifTrue: ["No filters"
    ^m_receiver _perform: m_selector withArguments: m_arguments].
"Only one argument signals are implemented"
"Apply filters"
^rFilter value:(m_receiver _perform: m_selector withArguments:
    (Array with: (fFilter value: (m_arguments at: 1)))).--
```

---

Please replace the paragraph beginning at page ~~22~~ 23, line ~~10~~ 33, with the following rewritten paragraph:

---

A20

--The current implementation keeps the information about channels distributed in the ***out*** variable 932, 934 of each connecton 908, 906 so the executive is only invoked to handle changes in the structure (i.e., message breaking). This design,

although it implements conceptually the role of the executive, might prove more efficient in distributed applications. Additionally, caching systems can also be become easier to implement. When the number of channels is very large, a central store implementation would need a hashing system to handle messages efficiently. Thus distributing channel information is an effective way to avoid hashing. In static structure networks, the message **doesNotUnderstand:** is not strictly necessary if some pre-processing is done before compilation, replacing calls to variable ***out*** with a defined message. In this case the exception mechanism would not be necessary and the implementation becomes faster. So the breaking process could be implemented as a pre-processor by those skilled in the art. The actual code of the message **doesNotUnderstand**: implemented in Smalltalk MT is as follows

A20

```
!ConnectonOutput 05:24 - 10/04/97!
doesNotUnderstand: aMessage
	|ms selector filter|
	selector := aMessage at: 2. "selector"
	ms := ~~extLinks~~ extChannels at: selector ifNone: nil.
	ms isNil ifTrue: [self error:'Gate: ',selector asString,'does not exist.'].
	filter := filters at: selector ifNone: nil.
	^self _broadcast: ms arguments: (aMessage at: 3) filter: filter.!!--
```

Please replace the paragraph beginning at page 25, line 30, with the following rewritten paragraph:

A21

--This method makes the separation between signals that are sent to connectons inside the network, and signals that are sent to connectons outside the network. In the later case, gate names are translated using the link information of the parent network. Variable *networkOutput* 910 is used to send messages from a network to external connectons.--

Please replace the paragraph beginning at page 25, line 34, with the following rewritten paragraph:

A22

--The call **_extMessage: aSCMessage arguments: anArray** is defined in the network output to access connectons outside the network, and is defined by

**!NetworkOutput 09:04-10/04/97!**
**_extMessage: aSCMessage arguments: anArray**

A22

```
|ms selector filter|
selector := aSCMessage selector.
ms := extLinks extChannels at: selector.
ms isNil ifTrue: [self error: 'Port:',selector asString,'does not exist.'].
filter := filters at: selector ifNone: nil.
^self _broadcast: ms arguments: anArray filter: filter!!--
```

Please replace the paragraph beginning at page 26, line 1, with the following rewritten paragraph:

A23

--Networks also need to break messages. Although messages coming from a connecton within a network can be easily sent to the output directly, because they are already broken, messages arriving to the network from the outside must also be identified for they can come from a standard calling mechanism, that is, the message

**connecton message**

must work whenever connecton represents a basic connecton or the connecton is an ensemble. The message breaker in the network is defined by

**!NetworkOutput. CALLBACK methods 09:04-10/04/97!**
**doesNotUnderstand: aMessage**

```
|ms selector filter|
selector := aMessage at: 2. "selector"
ms := intLinks intChannels at: selector ifNone: nil.
ms isNil ifTrue: [self error:'Gate: ',selector asString, 'does not exist'].
filter := inFilters at: selector ifNone: nil.
^self _broadcast: ms arguments: (aMessage at: 3) filter: filter.!!--
```

Please replace the paragraph beginning at page 26, line 18, with the following rewritten paragraph:

A24

--If the network is an inner model and it is not accessed by regular messages, signals are transmitted by the method **_perform:withArguments:** defined by

**!NetworkOutput 09:04-10/04/97!**
**_perform: selector withArguments: arguments**

```
|ms filter|
ms := intLinks intChannels at selector ifNone: nil.
ms isNil ifTrue: [self error.'Gate:',selector asString,'does no exist'].
filter := inFilters at: selector ifNone: nil.
^self _broadcast: ms arguments: arguments filter: filter.!!--
```

Please replace the paragraph beginning at page 28, line 23, with the following rewritten paragraph:

A25

--Structural inheritance can be used for several purposes. For example, a connecton superclass can be used to provide the common structure to its connecton subclasses. These connecton subclasses therefore must only just add the structural differences. We illustrate the use of inheritance to create a NOR4 gate (Fig. 13) from an OR4 gate (Fig. 11 ~~9~~). Fig. 11 represents the ~~The~~ OR4 gate 1102, which includes 4 inputs **in1** 1112, **in2** 1114, **in3** 1116 and **in4** 1118, and one output gate **out** 1120. ~~(represented in Fig. 11), is a 4 input gate and is~~ The OR4 gate 1102 is a network made of 3 basic two input OR gates OR1 1104, OR2 1106 and OR3 1108 and an OR4 executive 1110. The definition of the OR4 connecton class in given Fig. 12, where the definition of the network is made within the connecton Structure method.--

Please replace the paragraph beginning at page 28, line 31, with the following rewritten paragraph:

A26

--From the OR4 connecton class we can easily derive the NOR4 class depicted as NOR4 gate 1302 in Fig. 13. The DESMOS definition is given in Fig. 14, where the method **connectonStructure** only needs to define the differences from the initial OR4 gate 1102 depicted in Fig. 11. In Fig. 13, the NOR4 gate 1302 includes the OR gates 1104, 1006 and 1108, the NOR4 executive 1306 and the NOT gate 1304. In the definition of Fig. 14, the ~~The~~ first line obtains the structure of the parent class. The second line adds a model named #NOT of connecton class NOT. Actual ~~instances~~ connectons are obtained by replacing aliases by connecton class instances. ~~, that is~~ In this case, symbol #NOT is replaced by an instance of connecton class NOT named #NOT.--

Please replace the paragraph beginning at page ~~28~~ 29, line ~~31~~ 4, with the following rewritten paragraph:

A27

--We now describe how the Smalltalk MVC framework can be implemented using connectons, see S. Lewis. *The Art and Science of Smalltalk.* Prentice Hall, 1995. We use a simple graphical system 1502 represented in Fig. 15, which displays the speed of a car. The transducer connecton 1508 receives at gate **apperature:** 1514

~~converts~~ the throttle aperture from the throttle connecton 1506 that send the throttle aperture via gate **value:** 1512. The transducer connecton 1508 then coverts the aperture throttle to a value of velocity, and this value is sent to the output gate **speed:** 1518. The level meter connecton 1510 receives the velocity at gate **value:** 1520 and displays the speed of the car. It is noted that the transducer connecton 1508 includes a gate **x:y** 1516 that is not linked to any other connecton. The transducer operates independently of the connectons linked to its output. In the system of Fig. 16, two graphical ~~widgets~~ widget connectons 1604 and 1608 that display the value of the velocity and position have been added. The system of Fig. 16 also includes a control executive 1604. A plot chart connecton 1608 has been added and linked through the establishment of a new channel between the gate **speed:** 1518 of the transducer 1508 and the input gate **sample:** 1610 of the plot chart 1608. An XY-Plot connecton 1604 was also added to draw car position over time. The input gate **x:y:** 1606 of the XY-Plot chart 1604 is linked from the output gate **x:y:** 1516 of the transducer 1508. The **aperture:** action of the transducer is defined by

A27

**aperture: aValue**

|speed position|
speed := self computeSpeed: aValue.
position := self computeSpeed: aValue.
***out*** speed: speed.
***out*** x: (position x) y: (position y).--

---

Please replace the paragraph beginning at page ~~28~~ 31, line ~~31~~ 26, with the following rewritten paragraph:

---

--To show the power of the Connecton paradigm, we now describe an example taken from the software design pattern approach and compare it with the corresponding Connecton solution. Most of the so called structural patterns described in, E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. and remarkably, some of the behavior patterns can be replaced by connectons without the introduction of additional elements, or a simplification of the problem. The Chain of Responsibility pattern, categorized as a behavioral pattern, is a framework to pass commands through a chain of objects. Each object in the chain has two possibilities: to accept the command or to pass it

A28

A28

to the next element in the chain if the command cannot be locally handled. Each pattern is based upon a set of special purpose classes. The connecton solution, depicted in Fig. 17, does not demand any special model to handle chains. A chain is just a special topology, that is, a chain is just a specific problem of Connecton composition. The chain connecton 1702 depicted in Fig. 17 includes an executive connecton 1704, connectons A 1706, B 1708 and C 1710. The chain connecton 1702 includes a gate **in:** 1712 and a gate **out:** 1714. Each of the connectons 1706, 1708 and 1710 includes a gate **in:** 1716 and a gate **out:** 1718.--